International Journal of Digital Information and Wireless Communications (IJDIWC) 1(2): 545-553
The Society of Digital Information and Wireless Communications, 2011(ISSN 2225-658X)

From Relational Databases to XML Documents: Efficient alternatives for publishing

Mihai Stancu
Faculty of Mathematics and Computer Science,
Department of Computer Science,
University of Craiova, Romania
mihai.stancu@yahoo.com

ABSTRACT

XML language tends to become a standard for data exchanges between various Web applications. However, despite the big extent of the XML language, most of this applications store their information in relational databases. This fact is unlikely to be changed considering many advantages of the relational database systems like: fiability, scalability, performance and working tools. Thus, while XML language is under development, the necessity of some mechanism to publish XML documents from relational databases is obvious.

KEYWORDS

XML, databases, publishing

1 INTRODUCTION

Currently, the main research issues related to XML documents are: XML documents publishing, validity and typechecking of published XML documents and storing the XML documents. In this paper we will consider the issue of publishing XML documents.

The process of publishing relational data in XML documents needs two requirements to be accomplished. First requirement is a language that can specify the conversion beetwen relational data and XML documents. The second requirement refers to an efficient implementation of the conversion mechanism. The language specification describe the way that the records from one or more tables are structured and tagged to the hierarchical XML document. Considering a conversion language of this kind, an efficient implementation of the conversion mechanism raises many problems: the relational information is flat, while XML documents are hierarchical and tagged. In this paper we will analyze different variants of implementations for publishing XML documents.

The rest of this paper is organized as follows. Section 2 provides a short overview of XML language and defines a transformation language specification. Section 3 presents the space of alternatives for XML publishing and a comparison of these alternatives. The section 4 describes node outer union approach. Section 5 analyzes some potential problems that can arrive when implementing node outer union approach and provides a usefull way to avoid them. Section 6 presents some conclusions and future work directions.

2 A GENERIC TRANSFORMATION LANGUAGE SPECIFICATION

Extensible Markup Language (XML) [8] is a hierarchical data representation standard for information exchange in

Web. Many industry proposals [11] standardize document type descriptors (DTDs) [8], which are basically schemas for XML documents. These DTDs are being developed for domains as diverse as Business Solutions, Outsourcing [10] and Manufacturing [12]. Other alternative to DTDs is XMLSchema [13] that provides a detailed way to define data constraints. An XML document is a structure of nested elements, starting from one root element. Each element has an associated tag and can have attributes with associated values and/or nested elements. Figure 1 represents an XML document that store information for a web application in a version management tool.

```
<application id="C1">
    <name> CaseManagement </name>
    <versions>
        <version id="CN1"> 1 </version>
        <version id="CN2"> 2 </version>
    </versions>
    < modules>
        < module id="CE1" vers="CN2">
            <date> 1 January 2006 </date>
            <controllers>
                <controller id="I1"> Dispatcher controller </controller>
                <controller id="I2"> Download controller </controller>
            </controllers>
            <views>
                <view id="P1"> /pages/login.jspx </ view>
                <view id="P2"> /pages/home.jspx </ view>
                <view id="P3"> /pages/download.jspx </view>
            </views>
        </module>
        < module id="CE2" vers="CN1">
            ...
        </module>
    </modules>
</ application>
```

Figure 1. An XML document that describes an application

The application is represented by <application> element that is the document's root. The application has an *id* attribute with C1 value. The *id* attribute is a special one that has the role to uniqely identify another element from the XML document. Each application has a name denote by <name> subelement of the application element. Also, application element has subelements for his versions and modules. More details on XML can be found in [7].

We need a generic transformation language to define the conversion from relational data to XML documents. We will consider a SQL-based language in witch nested SQL clauses are used to model the imbrication and SQL functions are used for XML element construction [4].

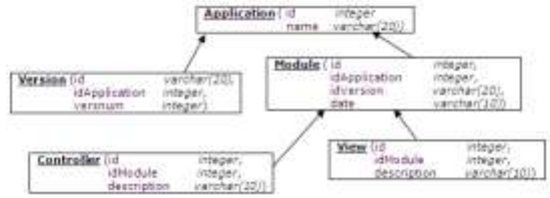


Figure 2. Application relational schema

Capitalize the first letter of each word.

We should consider the relational schema in Figure 2 that models the application information in Figure 1 in a relational form. The tables considered are: *Application, Version, Module, Controller* and *View*. Each table contains an id field that is a primary key and the relations between the tables are accomplished by foreign keys identifiable by arrows. In order to convert data in this relational schema to the XML document in Figure 1, we can write an SQL query which follows the nested structure of the document, as there can be seen in Figure 3.

```
Select app.name APP(app.id, app.name
        ( Select XMLAGG(VERS(vers.id, vers.versnum))
          From Version vers
          Where vers.idApplication = app.id),
        ( Select XMLAGG(MODL(modl.id, modl.idVersion, modl.date,
                ( Select XMLAGG(CONTROLLER(controller.id, controller.description))
                  From Controller controller
                  Where controller.idModule = modl.id),
                ( Select XMLAGG(VIEW(view.id, view.description))
                  From View view
                  Where view.idModule = modl.id)))
              group order by modl.date
          From Module modl
          Where modl.idApplication = app.id))
From Application app
```

Figure 3. SQL query to construct XML documents from relational data

The query in Figure 3 produces both SQL and XML data. Each resulted tuple contains the name of the application together with the XML representation of the application. The entire query consists of a number of correlated sub-queries. Analysing the query from top to bottom we can see that on the first level, the highest one, there is the query that elicits each application from the Application table. For each application, there are used correlated sub-queries in order to retrieve the application's versions (lines 02-04), respectively the modules (lines 05-14).

The definition of XML APP constructor can be seen in Figure 4. Conceptually, it is perceived as a scalar function that returns an XML. For each tuple, it tags the specific columns producing thus an XML fragment.

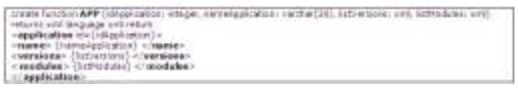


Figure 4. An XML constructor definition

The other correlated sub-queries can be similarly interpreted with the VERS, MODL, CONTROLLER and VIEW constructors, analogous to APP. This way each nested query returns an XML fragment. The XMLAGG aggregate function is used to concatenate these XML fragments produced by the constructors.

To order XML fragments, the XMLAGG aggregate function must get ordered inputs. For example, to order the modules achieved by a application chronologically we must make sure that the XMLAGG function from line 05 of Figure 3 concatenates the modules in this exact order. Because the ordered inputs for aggregation functions are not born by the SQL standard, extensions of the SQL language that tolerate this facility can be used. Such an extension can be the introduction of the *group order by* clause as there can be seen in Figure 3 line 12, to chronologically order a application's modules before they are introduces in the aggregate XMLAGG function.

3 VARIANTS OF IMPLEMENTATION

To be able to analyse the alternatives for publishing relational data as XML documents we have to take into account the main difference between the relational tables and the XML documents which means that XML documents, unlike the relational tables, have tags and nested structure. Therefore, somewhere along the process of converting relational tables to XML documents, the tags and the structure must be added. We can thus distinguish several possibilities of approaching and that is the adding of tags during the final step of the query processing (late-tagging) or by doing this at the beginning of the process (early-tagging). Similarly, the structure adding can be done as the final step of processing (late-structuring) or it can be done earlier (early-structuring) [4]. These two dimensions of tagging and structuring gave rise to a space of processing alternatives represented in Figure 5.

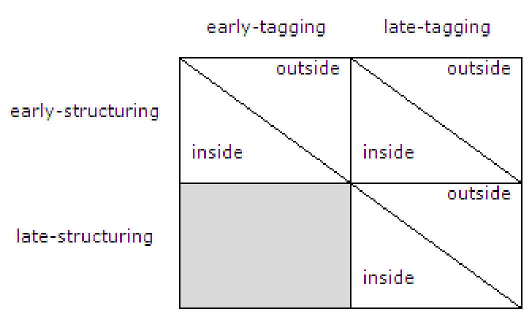


Figure 5. Space of alternatives for XML documents publishing [4]

All these alternatives have variants that depend on accomplishing these operations inside or outside the SGBD. We should see that inside refers to the fact that tagging and structuring are completely accomplished inside the relational engine, while outside, the fact that these operations are done totally outside the relational engine. There can also be observed that early-tagging and late-structuring is not a viable variant because the tagging of an XML document cannot be done without already having its structure.

Comparative studies of these alternatives [4] show the advantages and disadvantages for each one and when it is useful to choose one or another alternative. The main characteristics have been summarized in Figure 6.
The qualitative assessments indicate that every alternative has some potential disadvantage. Taking into account the various implementations of the presented approaches, we can draw the following conclusions:
1. Building XML documents inside the relational system is more efficient than outside it
2. When processing can be done only by using the main memory, the unsorted outer union approach is stable always among the best (inside or outside)
3. When processing cannot be done just by using the main memory, the sorted outer union approach (inside or outside) is the choice. This is due to the fact that the relational sorting operator scales very easily.
Performance evaluation of the alternatives was done [4] in order to determine which ones are likely to win in practice (and in what situations). Seen as a whole, the main disadvantages of the outer union approaches are not significant. They refer to the fact that tuples of large sizes are created but with many null values. The efficient methods of compressing null values can considerably reduce the size of the tuples appeared as a result of outer union.

Classification		Approach	Description	Potential problems
Early-structuring Early-tagging	Outside	Stored procedure	Perform separate queries conform to document's structure by doing iterative nested joins outside the relational engine.	1) Many SQL queries 2) Fixed join strategy (iterative nested joins)
	Inside	correlated CLOB	Internal variant of the stored procedure approach. Use CLOB fields to construct intermediate XML fragments.	1) Fixed join strategy (iterative nested joins). 2) Many intermediate CLOB fields created during query processing
	Inside	De-correlated CLOB	De-correlated version of the correlated CLOB approach. Use also intermediate CLOB fields	1) Many intermediate CLOB fields created during query processing
Late-structuring Late-tagging	Inside or outside	Redundant relation	Create a relation with redundancy because each child of a parent is repeated many times.	1) Data redundancy. 2) Memory overflow in the hash-based tagging mechanism.
	Inside or outside	Unsorted path outer union	Create outer union of the leaf elements to avoid redundancy. Use a hash-based tagging mechanism.	1) Data redundancy (on a smaller scale) 2) Memory overflow in the hash-based tagging mechanism. 3) Wide tuples.
	Inside or outside	Unsorted node outer union.	Similar to unsorted path outer union, but include also tuples for non-leaf elements in outer union.	1) Memory overflow in the hash-based tagging mechanism. 2) Wide tuples.
Early-structuring Late-tagging	Inside or outside	Sorted path outer union	Structure the results of unsorted path outer union by sorting these results in document order.	1) Data redundancy (on a smaller scale). 2) Wide tuples. 3) Require total order of Relational result.
	Inside or outside	Sorted node outer union	Structure the results of unsorted node outer union by sorting these results in document order.	1) Wide tuples. 2) Require total order of Relational result.

Figure 6. Approaches for publishing XML documents

4 LATE-STRUCTURING, LATE-TAGGING

In this class of alternatives that delay tagging and structuring, these operations are done as the last step of the XML building process. So, in constructing an XML document there are two stages: (a) content creation, where the relational data is produced and (b) tagging and structuring, where the relational data is tagged and structured to produce an XML document.

4.1 Unsorted node outer union approach

The outer union approach of the path eliminates much from the content and processing redundancy. This is due to the fact that parents and children are represented in separate tuples. Anyway, there is still some redundancy in the data because the parents information are replicated in each child. For example, information about the application such as name are replicated in each version that is associated to the application. One way of solving this problem is to send the information about the parent directly to the outer union, and the children (and all the descendents) retain only the id field of the parent. This option is called the node outer union approach.

Figure 7 indicates the execution plan for the node outer unification, and the corresponding SQL query is shown in Figure 8.

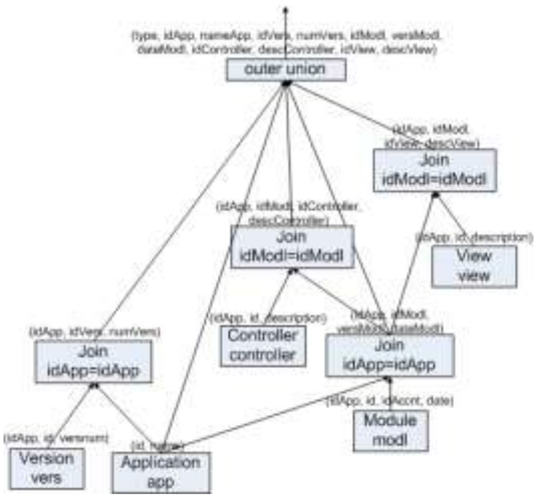


Figure 7. SQL query execution plan for the unsorted node outer union approach

```
-- We compute all paths from root to leafs
with app (idApp integer, nameApp varchar(20)) as (
    select app.id, app.name
    from Application app
),
versApp (idApp integer, idVers integer, numVers integer) as (
    select app.id, vers.id, vers.versnum
    from Version vers, app
    where vers.idApplication = app.id
),
modlApp (idApp integer, idModl integer, versionModl varchar(20), dateModl varchar(10)) as (
    select app.id, modl.id, modl.idVersion, modl.date
    from app, Module modl
    where app.id = modl.idApplication
),
controllerModlApp (idApp integer, idModl integer, idController integer, descController varchar(20)) as (
    select modlapp.idApp, modlapp.idModl, controller.id, controller.description
    from Controller controller, modlApp modlapp
    where controller.idModule = modlapp.idModl
),
viewModlApp (idApp integer, idModl integer, idView integer, descView varchar(20)) as (
    select modlapp.idApp, modlapp.idModl, view.id, view. description
    from modlApp modlapp, View view
    where modlapp.idModl = view.idModule
),
-- Main query that make node outer union
select 0, idApp, nameApp, null, null, null, null, null, null, null, null, null
from app
    union all
select 1, idApp, null, idVers, numVers, null, null, null, null, null, null, null
from versApp
    union all
select 2, idApp, null, null, null, idModl, versionModl, dateModl, null, null, null, null
from modlApp
    union all
select 3, idApp, null, null, null, idModl, null, null, idController, descController, null, null
from controllerModlApp
    union all
select 4, idApp, null, null, null, idModl, null, null, null, null, idView, descView
from viewModlApp
```

Figure 8. SQL query for the unsorted node outer union approach

There can be seen that all information about parents (for example referring to the application) are directly transmitted to the union operator, and in their descendents there are sent only those particular id fields. Because all information about parents is sent directly to the outer union operator, the use of a normal join is enough not to lose information.

The node outer union approach reduces the redundancy appeared in the outer union approach of the path because parents information are not replicated into their children. However, the node outer union approach increases the number of tuples in the result because each parent is represented by a separate tuple. A potential problem for both approaches based on outer union is that the number of columns in the result tuples increases together with the width and depth increase of the XML document. Even if a small number of these columns is used to store values in a tuple, without techniques of compressing the null values there can be registered an increase of processing because of the large measurements of the tuples.

4.2 Hash-based Tagger

The final step in the late-structuring, late-tagging class is to tag and structure the relational content to form the result. This can be done either inside or outside the relational engine. If it is performed inside the relational engine, it can be implemented as an aggregate function. This only aggregation function will make the operations carried on by all XML constructors and the aggregation functions from the user query. This will ensure us that there will be no large items retained along processing, which is a possible disadvantage for the CLOB approaches.

In order to structure and tag the results, inside or outside the engine, we need to reach two stages: (a) grouping under the same parents all brothers in the wanted XML document (and eliminating

duplicates in case of redundant relation approach) and (b) extracting information from each tuple and tagging them to produce the resulted XML document. An efficient way of grouping brothers is to use a main memory hash table to check the parent of a node by giving the type of the parent node and its id field value (including the id fields of the ancestors).
Every time a tuple that contains information about an XML document is observed, the type of element and the id values of its ancestors are looked for in the hash table, to find out if its parent is or is not present in the hash table. If the parent is found, a new XML element will be created and it will be added as son of this particular parent. The case when the parent is not present may frequently appear because the result tuples do not appear in a certain order. In this case a hash on its grandfather is performed and if it is found the space necessary for the parent is retained as son of the grandfather. If neither the grandfather is found the procedure is repeated for the ancestors of that element until the root element is reached.
After all the input tuples have been hashed, the entire tagged and structured result can be written out as an XML file. If a specific purchase order is required for the elements of the resulting XML document, such as a chronological ordering, then that purchase order can either be maintained as children are added to a parent or it can be enforced by a final sort before writing out the XML document.
The main limitation of using a hash-based tagger is that performance can decrease rapidly when there is insufficient memory to hold the hash table and the intermediate result.

5 EARLY-STRUCTURING, LATE-TAGGING

These alternatives try to diminuate the disadvantages from the late tagging and late structuring approaches that is the hash-based tagger that needs to perform complex memory management when memory is insufficient. To eliminate this problem, the relational engine can be used to produce “structured relational content”, which can then be tagged by using a constant space tagger.

5.1 Sorted node outer union approach

The key to structuring relational content is to order it the same way that it needs to appear in the result XML document. This can be achieved by ensuring that:
1. The information of any parent node appear before or along with any of his descendant’s information.
2. All the information of a parent node and his descendants appear successively in the sequence of tuples.
3. The information of a node of a certain type appear before the information of any of his seablings of other type.
4. The tuples order respect any other criteria defined by the user.
In order to ensure the following of the four conditions, we only need to sort the results of the node outer union approach on the id field and follow the criteria the user has defined [4], so that:
• the id field of a parent node comes before the id fields of his children
• the id field of a brother node comes in the reversed order of the brother nodes sorting in XML document;
• the sorting fields of a node defined by the user come right before the id node of that particular node.
Thus, in our example, sorting the result on the node outer union method will be

done on the sequence: (AppId, VersId, POId, ControllerId, ViewId). This way the results will be sorted in document order.

The query execution plan and the corresponding SQL query are shown in Figures 9, 10 and 11.

For the operations to be done correctly the user-defined sorting fields must be propagated from a parent node to the descending nodes before outer union as shown in Figure 9. We must also make sure that tuples having null values in the sort fields occur before tuples having non-null values for these fields (null-low sort). This is necessary for us to be sure that parents and brothers come in right order.

The sorted outer union approach has the advantage of scaling large data volumes because relational database sorting is disk-friendly. The approach can also ensure user-specified orderings with little additional cost. However, it does do more work than necessary, since a total order is produced when only a partial order is needed.

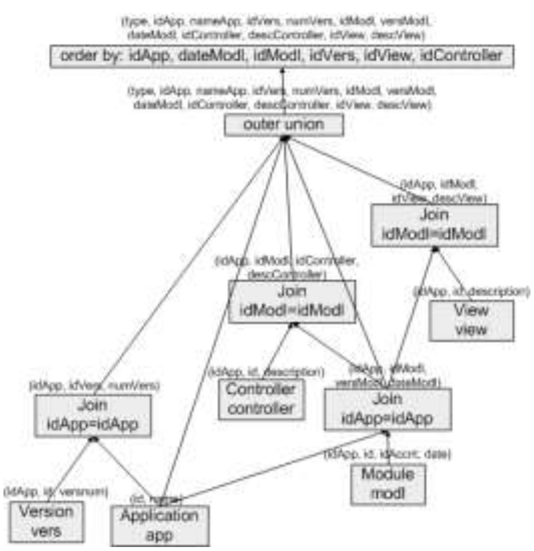


Figure 9. SQL query execution plan for the sorted node outer union approach

```
-- We compute all paths from root to leafs
01. with app (idApp integer, nameApp varchar(20)) as (
02.     select app.id, app.name
03.     from Application app
04. ),
05. versApp (idApp integer, idVers integer, numVers integer) as (
02.     select app.id, vers.id, vers.versnum
03.     from Version vers, app
04.     where vers.idApplication = app.id
05. ),
06. modlApp (idApp integer, idModl integer, versionModl varchar(20), dateModl varchar(10)) as (
07.     select app.id, modl.id, modl.idVersion, modl.date
08.     from app, Module modl
09.     where app.id = modl.idApplication
10. ),
11. controllerModlApp (idApp integer, idModl integer, idController integer, descController varchar(20)) as (
12.     select modlapp.idApp, modlapp.idModl, controller.id, controller.description
13.     from Controller controller, modlApp modlapp
14.     where controller.idModule = modlapp.idModl
15. ),
16. viewModlApp (idApp integer, idModl integer, idView integer, descView varchar(20)) as (
17.     select modlapp.idApp, modlapp.idModl, view.id, view.description
18.     from modlApp modlapp, View view
19.     where modlapp.idModl = view.idModule
20. ),
21. -- Main query that make node outer union
22. select 0, idApp, nameApp, null, null, null, null, null, null, null, null, null
23. from app
24.     union all
25. select 1, idApp, null, idVers, numVers, null, null, null, null, null, null, null
26. from versApp
27.     union all
28. select 2, idApp, null, null, null, idModl, versionModl, dateModl, null, null, null, null
29. from modlApp
30.     union all
31. select 3, idApp, null, null, null, idModl, null, null, idController, descController, null, null
32. from controllerModlApp
33.     union all
34. select 4, idApp, null, null, null, idModl, null, null, null, null, idView, descView
35. from viewModlApp
```

Figure 10. SQL query for the unsorted node outer union approach

```
-- The lines 1-38 compute node outer union same like in figure 8
01.
  //compute node outer union into "NOU" temporary table
38. );
39. -- Main query that sort the results of outer union
40. select type, idApp, nameApp, idVers, numVers, idModl, versionModl, dateModl, idController, descController,
idView, descView
41. from outerUnion
42. order by idApp, dateModl, idModl, idVers, idView, idController
```

Figure 11. SQL query for the sorted node outer union approach

5.2 Constant space tagger

Once structured content is created, the last step is to tag and construct the result XML document. Since tuples arrive in document order, they can be immediately tagged and written out. The tagger only requires memory to remember the id field of the last parent. These ids are used to detect when all the children of a particular parent node have been seen so that the closing tag associated with the parent can be written out. For example, after all the controllers and views of a module have been seen, the closing tag for module (</module>) has to be written out. To detect this, the tagger stores the id of the current module and compares it with that of the next tuple. The storage space required by the constant space tagger is proportional only to the level of nesting and is independent of the size of the XML document.

6 SAMPLE IMPLEMENTATION AND FINE-TUNING

Considering the conclusions from previouse section, we will try to implement XML documents publishing using node outer union approach. Choosing the sorted or unsorted variant depends by the size of the data volume that need's to be processed. Despite that memory allocation problems are permanently supported by hardware evolution, the high data volume needed by applications is growing in time so that the maintenance process of these applications can be problematic. In these conditions, the sorted node outer union approach can be a viable option.

In this sample implementation we use PostgreSQL 8.0 [9] as the relational engine and coding was done in Java (JDK 1.4.2_05) [14]. The database connection was made through JDBC driver postgresql-8.0.309.jdbc3.jar.

Considering Postgres specific SQL syntax, the implementation of the functions app, versApp, modlApp, controllerModlApp, viewModlApp and NOU from Figure 10, was done using VIEW elements (Figure 12).

One problem met when using PostgreSQL relational database engine is that the null values are sortes high (null-high sort). For this publishing approach, this fact is a drawback and it was avoided by adding an additional sorting criteria for each original sorting criteria. For instance, in order by clause, instead of idVers asc, we can use idVers not null, idVers asc. Thus, first sorting criteria will assure us that null values of the sorted column will appear first in the results and the second sorting criteria will assure us the effective order of the column.

The constant space tagger was implemented in Java. The document's generation is relatively simple, done by looping through the sorting results. Additional attention is needed when closing the elements tags for parents nodes. This thing was done using a stack data structure. After we run the constant space tagger, the generated applications.xml file is obtained and can be seen in Figure 15.

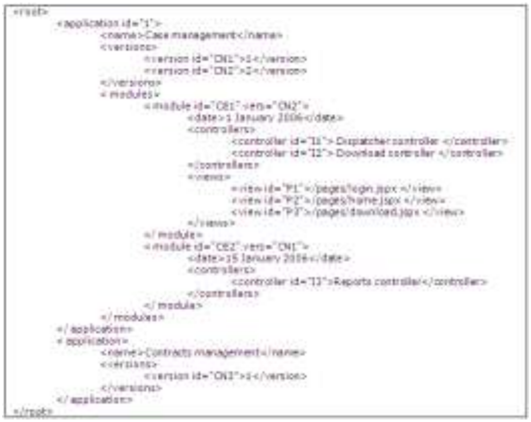
Figure 12. PostgreSQL VIEWS

The main query that sort the results of the node outer union is shown in figure 13 and the obtained results in Figure 14.

```
Select
    type, idapp, nameapp, idvers, numvers,
    idmodl, versionmodl, datemodl, idcontroller, desccontroller, idview, descview
from NOU
order by
    idapp is not null, idapp asc,
    datemodl is not null, datemodl asc, idmodl is not null, idmodl asc,
    idvers is not null, idvers asc,
    idview is not null, idview asc,
    idcontroller is not null, idcontroller asc
```

Figure 13. main SQL query

Figure 14. results of the main SQL query



```
<root>
    <application id="1">
        <name>Case management</name>
        <versions>
            <version id="CN1">1</version>
            <version id="CN2">2</version>
        </versions>
        <modules>
            <module id="CE1" vers="CN2">
                <date>1 January 2006</date>
                <controllers>
                    <controller id="I1"> Dispatcher controller </controller>
                    <controller id="I2"> Download controller </controller>
                </controllers>
                <views>
                    <view id="P1"> /pages/login.jspx </view>
                    <view id="P2"> /pages/home.jspx </view>
                    <view id="P3"> /pages/download.jspx </view>
                </views>
            </module>
            <module id="CE2" vers="CN1">
                <date>15 January 2006</date>
                <controllers>
                    <controller id="I3">Reports controller</controller>
                </controllers>
            </module>
        </modules>
    </application>
    <application>
        <name>Contracts management</name>
        <versions>
            <version id="CN3">1</version>
        </versions>
    </application>
</root>
```

Figure 15. applications.xml document

7 CONCLUSIONS AND FUTURE WORK

In this paper we have presented various alternatives for XML publishing from relational database systems and we focus on a specific alternative. We analyzed potential problems that can rise for node outer union implementation and we found some way to avoid them.

The future work can include various possibilities as the impact of parallelism, the addition of new operators inside the relational database engine to increase the performance of the outer union execution, and the design of efficient memory management techniques for unsorted outer union approach.

8 REFERENCES

1. Suciu, D.: On Database Theory and XML, SIGMOD Record, vol. 30, no. 3, (2001).
2. Fernandez, M., Kadiyska, Y., Suciu, D., Morishima, A., Tan, W.: SilkRoute: A Framework for Publishing Relational Data in XML, ACM Transactions on Database Systems, vol. 27, no. 4, (2002), pp. 438-493.
3. Carey, M., J., Florescu, D., Ives, Z., G., Lu, Y., Shanmugasundaram, J., Shekita, E. J., Subramanian, S., N., XPERANTO: Publishing Object- Relational Data as XML, International Workshop on the Web and Databases, (2000).
4. Jayavel, S., Eugene S., Rimon B., Michael C., Bruce L., Hamid P., Berthold R.: Effciently publishing relational data as XML documents, The VLDB Journal, (2001).
5. Seshadri P., Pirahesh H., Leung T.Y.C., Complex Query Decorrelation, Proc. International Conference on Data Engineering (ICDE), La., USA, (1996).
6. Fagin R., Multi-valued Dependencies and a New Normal Form for Relational Databases, ACM Transactions on Database Systems, 2(3), (1977).
7. Rusty H., W. Scott M., Xml in a Nutshell, O'Reilly & Associates, Inc., (2004).
8. W3C - World Wide Web Consortium, Extensible Markup Language (XML) 1.1 (2nd Edn), W3C Recommendation, (2006), http://www.w3.org/TR/xml11/.
9. PostgreSQL, PostgreSQL - Documentation, http://www.postgresql.org/docs/
10. LionBridge, http://en-us.lionbridge.com/Default.aspx?LangType=1033
11. Cover R., The XML Cover Pages, http://xml.coverpages.org/.
12. PLUS Vision Corp., http://www.plus-vision.com/.
13. W3C - World Wide Web Consortium, XML Schema, W3C Candidate Recommendation, (2004), http://www.w3.org/TR/xmlschema-2/.
14. Oracle, Java Platform - Standard Edition, http://www.oracle.com/technetwork/java/javase/overview/index.html